Andrew Kraft

andrewlkraft@ucsb.edu

3410933

MP1 Report

Architecture:

My naive bayes classifier takes input from the training file and stores it in a vector of documents. Each document within this vector contains a boolean rating of true for a positive review and false for a negative review, as well as a size variable representing the number of words in the document and a hash table relating each word to the number of times it appears in that document. Using this vector of documents, I created another hash table which relates words to their probabilities of appearing in a positive and negative review. To ease the creation of this map, I created another hash table which maps a string to the number of times it appears in a positive or negative review. These tables will be referred to as *pTable* and *dTable* respectively. Hash tables are used for their O(1) lookup time and because the order of these elements is unimportant, since we will always be performing lookups by word value.

Preprocessing:

Nothing fancy is done to the data from the file itself, the code simply reads the file line by line, first checking the value at the end of each line to see whether the array is positive or negative, then erasing the last two characters of each line, which should be the comma and review number. After cleaning the string like this, it transforms the sentence into a document with individual words, and appends the document to the document vector.

Once the document vector is created, the program uses it to create *dTable*, iterating through the entire vector and adding each word in the corpus to *dTable*, while incrementing the counts in *dTable*. Once *dTable* has been created, its values are used to create *pTable*, using a probability calculation method described in the next section.

Model Building:

The naive bayes model used in this classifier is quite simple, using a log model to add probabilities instead of having to multiply them and deal with very small numbers. When calculating the log model, the network makes the assumption that p(positive review) = p(negative review) = 0.5, so it does not have to add these probabilities into the model since their values will be counted equally when the positive and negative probabilities are compared at the end to decide the label. For the same reason, no change is made to the probabilities when the classifier encounters a word outside of the model, since it assumes the probability of the review being positive or negative given that word is the same. When calculating the probabilities themselves, the model uses slight smoothing, using the equation:

$$p(Word = w_i | \, review = c_j) \ = \ \frac{N(Word{=}w_i, \, review{=}c_j) + \alpha}{N(review{=}c_j) + \alpha|Vocabulary|}$$

This equation smooths the values so that if the word doesn't appear very often in a certain type of review, and there are few words in reviews of that type, the probability of that word showing up is set equal to 1 / |Vocabulary|, but if a word does appear often in a review and there are many words from reviews of that type, the probability of that word showing up is closer to p(word and review) / p(review). I played around with the alpha value a bit, but couldn't find anything that worked better for the testing set than an alpha of 1, so for now alpha will just be equal to 1, though it could possibly be tuned to change the effect of this smoothing.

Results:

The classifier takes approximately 4 seconds to train and 4 to label, and achieves an accuracy score of around 0.89 for training and 0.85 for testing. To find the most predictive words for each class of review, I took the difference of log(p(word|pos)) and log(p(word|neg)) as the significance score, which is approximately equal to the difference in orders of magnitude of the two probabilities. Of all the words in the corpus, the ten most predictive of a positive review were “excelente”, “chibi”, “blox”, “sigil”, “producto”, “thottam”, “jameson”, “galciv”, and “tiempo”, while the most predictive for a negative review were “tera”, “refund”, “soe”, “rmah”, “dumbest”, “numbingly”, “deadliest”, “starkiller”, “garbage”, and “ros”.

Challenges:

One challenge I encountered was when I tried to implement tf-idf as a means of making neutral and often used words have less weight in labeling a review. After doing lots of research into tf-idf, and putting time into creating a function to transform the weights of words in the document vector to tf-idf form, the resulting accuracy of the network was significantly less than with normal count-based weighting. I realized that, though neutral words which appear often may not have direct positive or negative connotations, their usage could still suggest something about the review that the computer notices but I don’t.

Weaknesses

This is a very simple classifier, and therefore, it could be improved in many ways. I think one of the ways its performance could be improved most significantly is by using bigram terms as features, because it would add significantly more features and allow the classifier to recognise sequences of multiple words. One example of this is that both “Jameson” and “thottam” are very high in terms of predictability, but looking through the reviews, I found that these terms are related and both always appear next to each other. With a bigram setup, the classifier could recognize this and adjust weights accordingly. I also noticed that many of the most predictive words in either direction were words that appeared many times in a small number of reviews, and that the reviews they appeared in tended to be longer than average. I could use some text-size normalizer to reduce the impact of the words used in long reviews.